Announcements

- Will be posted no later than Monday
  - HW-01
  - PR-01

- Team formation
  - We can take a few minutes to do it now
  - Make private Piazza post with UBIT username and corresponding GitHub username.

- Team meetings
  - Once teams are formed we can schedule meeting times
Phases of a compiler

Figure 1.6, page 5 of text
Formally, a grammar $G = (N, \Sigma, P, S)$ is defined by 4 items:

1. $N$, a set of non-terminals
   $N = \{ X, Y \}$

2. $\Sigma$, a set of terminals (alphabet)
   $\Sigma = \{ a, b \}$ — for example
   $N \cap \Sigma = \{\}$ — general grammar constraints

3. $P$, a set of productions of the form (right linear)
   $X \rightarrow aY$
   $Y \rightarrow bX$
   $Y \rightarrow a$ — a right linear grammar describing a regular language
   $X \rightarrow \varepsilon$
   $X \in N, Y \in N, a \in \Sigma, \varepsilon$ denotes the empty string

4. $S$, a start symbol
   $S = Y$
   $S \in N$
languages & grammars

$L(G)$ is the set of all strings derivable from $G$ starting with the start symbol; i.e. it denotes the language of $G$. 
languages & grammars

Given a grammar $G$ the language it generates, $L(G)$, is unique.

Given a language $L$ there are many grammars $H$ such that $L(H) = L$. 
Given a grammar $G$ the language it generates, $\mathcal{L}(G)$, is unique. Given a language $L$ there are many grammars $H$ such that $\mathcal{L}(H) = L$.

Think about what this means for us: there is no single "correct" grammar for a language.

In fact, grammars for users vs. tool writers vs. compiler writers can all be different.

Given a language $L$ there are many grammars $H$ such that $\mathcal{L}(H) = L$. 

Lexical Analysis

- Lexical structure described by regular grammar
- Deterministic finite state machine performs analysis
How is a regular language defined?

- Recall that a language is a set of strings. This set can be finite or infinite.

- The possible regular languages over a given alphabet are defined inductively - construction given on next two slides.
LANGUAGE operations

base cases

- \{ \varepsilon \} is a regular language
- \forall a \in \Sigma, \{ a \} is a regular language

Recall, \varepsilon is the empty string
LANGUAGE operations

If \( L \) and \( M \) are regular, so are:

- \( L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \) \hspace{1cm} \text{UNION}
- \( LM = \{ st \mid s \in L \text{ and } t \in M \} \) \hspace{1cm} \text{CONCATENATION}
- \( L^* = \bigcup_{i=0}^{\infty} L^i \) \hspace{1cm} \text{KLEENE CLOSURE}
- \text{No other languages are regular}

\( L^i \) is \( L \) concatenated with itself \( i \) times:
- \( L^0 = \{ \varepsilon \} \), by definition
- \( L^1 = L \)
- \( L^2 = LL \)
- \( L^3 = LLL \), etc.
- \( L^* \) is the union of all these sets!
Example of L^* 

Suppose L is \{a, bb\}

L^0 = \{\varepsilon\}, by definition

L^1 = L = \{a, bb\}

L^2 = LL = \{aa, abb, bba, bbbb\}

L^3 = LLL = \{ aaa, aabb, abba, abbbb,
                  bbaa, bbabb, bbbba, bbbbbbb\}

L^4 =

...and so so...

L^* = \bigcup_{i=0,\infty} L^i = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa,
                                  aabb, abba, abbbb, bbba, bbbba, bbbaa, bbabb,
                                  bbbba, bbbbbbb, abbbb, bbabb, ... \}
Some regular languages over $\Sigma = \{0,1\}$

The base cases yield these regular languages:

$\{\varepsilon\}, \{0\}, \{1\}$

The inductive cases yield many more. Some are:

$\{0, 1\}, \{01\}, \{10\}, \{01, 10\}, \{0, 01\}, \{1, 01\}, \{0, 10\}, \{1, 10\}, \{0, 1, 01\}, \{0, 1, 10\}, \{0, 01, 10\}, \{1, 01, 10\}, \{00\}, \{000\}, \{0000\}, \{11\}, \{111\}, \{1111\}$, and many many more.

Can you demonstrate how each of these is regular?
Why use grammars?

- Recall that a language is a possibly infinite set of strings.
- A grammar gives us a way to describe, using finite means, an infinite set.
- Regular expressions are equivalent to regular grammars in expressive power: both regular grammars and regular expressions describe regular languages.
- If $X$ is a regular expression, $\mathcal{L}(X)$ denotes the set of strings recognized by $X$. 
Inductive definition of REGular EXpressions (regex) over a given alphabet $\Sigma$

$\epsilon$ is a regex

$\mathcal{L}(\epsilon) = \{\epsilon\}$

For each $a \in \Sigma$, $a$ is a regex

$\mathcal{L}(a) = \{a\}$
Regular expressions (regex)  
Inductive definition

Assume \( r \) and \( s \) are regexes.

- \( r|s \) is a regex denoting \( L(r) \cup L(s) \)
- \( rs \) is a regex denoting \( L(r)L(s) \)
- \( r^* \) is a regex denoting \( (L(r))^* \)
- \( (r) \) is a regex denoting \( L(r) \)

**Precedence:** Kleene closure > concatenation > union

**Associativity:** all left-associative (minimize use of parentheses: \( (r|s)|t = r|s|t \) )
Algebraic Laws

Assume \( r \) and \( s \) are regexes.

**Commutativity** \( r|s = s|r \)

**Associativity** \( r|(s|t) = (r|s)|t \) and \( r(st) = (rs)t \)

**Distributivity** \( r(s|t) = rs|rt \) and \( (s|t)r = sr|tr \)

**Identity** \( \varepsilon r = r \varepsilon = r \)

**Idempotency** \( r** = r^* \)
We can describe a regular language using a regular expression
Why do we care?

- We will be using a tool called FLEX to construct a lexical analyzer (a lexer) for the programming language we're constructing a compiler for.

- If we give FLEX a regular expression describing the lexical structure of our language, FLEX will produce a C program which acts as our lexer.

- The next step for us to understand (at a high level) how FLEX converts a regex to a C program.
A regular expression can be implemented using a finite state machine.

Finite state machines can be deterministic or non-deterministic:

- **DFA**
  deterministic finite automaton

- **NFA**
  non-deterministic finite automaton
Process of building lexical analyzer

1) spell out the language
Process of building lexical analyzer

2) formulate a regular expression
Process of building lexical analyzer

3) build an NFA
Process of building lexical analyzer

4) transform NFA to DFA
Process of building lexical analyzer

5) transform DFA to a minimal DFA
5) The minimal DFA is our lexical analyzer

FLEX generates a C program which implements the minimized DFA
Step 1: Construct NFA from regex
Nondeterministic Finite Automata (NFA)

- A finite set of states $S$
- An alphabet $\Sigma$, $\varepsilon \notin \Sigma$
- $\delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{P}(S)$ (transition function)
- $s_0 \in S$ (a single start state)
- $F \subseteq S$ (a set of final or accepting states)
Deterministic Finite Automata (DFA)

- A finite set of states $S$
- An alphabet $\Sigma$, $\varepsilon \notin \Sigma$
- $\delta \subseteq S \times \Sigma \times S$ (transition function)
- $s_0 \in S$ (a single start state)
- $F \subseteq S$ (a set of final or accepting states)
NFA vs DFA

Transition function

\[ \delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times P(S) \]

- No \( \varepsilon \)-transitions
- No multiple transitions
A state is a circle with its state number written inside.
Initial state has an arrow from nowhere pointing in. State 0 is often the initial state.
A final (accepting) state is drawn with a double circle.
Arrows are labeled with $\varepsilon$ ... 

$\varepsilon$  

... or $a \in \Sigma$. 

$a$  

for each $a \in \Sigma$
Regex → NFA

for each $a \in \Sigma$
Regex $\to$ NFA

$S^*$

$\varepsilon$

$N(s)$
Simple example

static
Simple example

static
Simple example

static
struct
Step 2: Construct DFA equivalent to NFA

Next time!
Step 3:
DFA minimization

Next time!